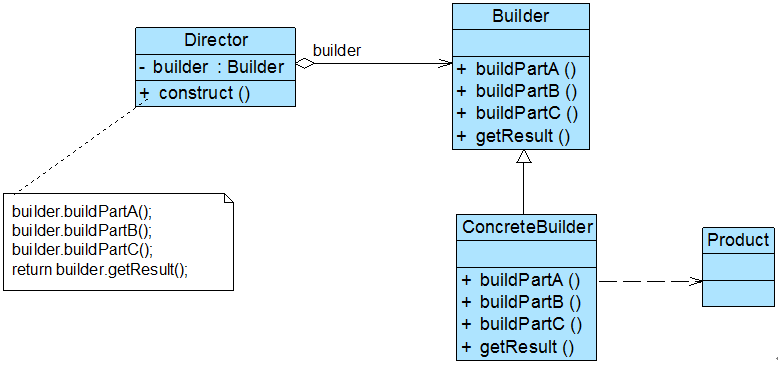
建造者模式又称为**生成器模式**，它是一种较为复杂、使用频率也相对较低的创建型模式。建造者模式为客户端返回的不是一个简单的产品，而是一个由多个部件组成的复杂产品。

建造者模式(Builder Pattern)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。建造者模式是一种对象创建型模式。

建造者模式结构如图所示：



● **Builder（抽象建造者）**：它为创建一个产品Product对象的各个部件指定抽象接口，在该接口中一般声明两类方法，一类方法是buildPartX()，它们用于**创建复杂对象的各个部件**；另一类方法是getResult()，它们用于**返回复杂对象**。Builder既可以是抽象类，也可以是接口。

● **ConcreteBuilder（具体建造者）**：它实现了Builder接口，实现各个部件的具体构造和装配方法，定义并明确它所创建的复杂对象，也可以提供一个方法返回创建好的复杂产品对象。

● **Product（产品角色）**：它是被构建的**复杂对象**，包含多个组成部件，具体建造者创建该产品的内部表示并定义它的装配过程。

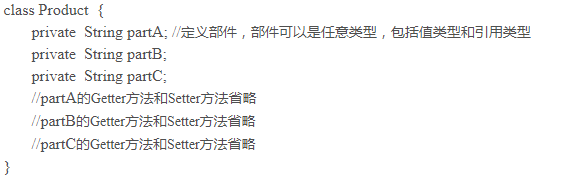
● **Director（指挥者）**：指挥者又称为导演类，它负责安排复杂对象的建造次序，指挥者与抽象建造者之间存在关联关系，可以在其construct()建造方法中调用建造者对象的部件构造与装配方法，完成复杂对象的建造。客户端一般只需要与指挥者进行交互，在客户端确定具体建造者的类型，并实例化具体建造者对象（也可以通过配置文件和反射机制），然后通过指挥者类的构造函数或者Setter方法将该对象传入指挥者类中。

复杂对象：

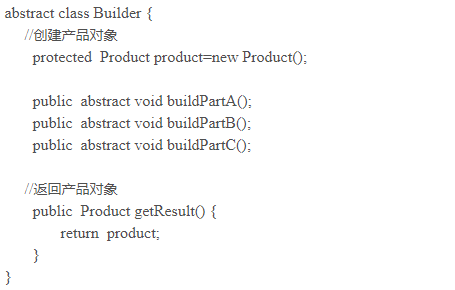
是指那些包含多个成员属性的对象，这些成员属性也称为部件或零件，如汽车包括方向盘、发动机、轮胎等部件，电子邮件包括发件人、收件人、主题、内容、附件等部件。

代码实现：

**产品角色类**（复杂对象）



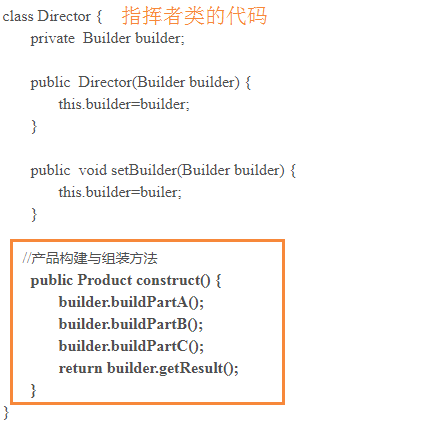
在**抽象建造者类**中定义了产品的创建方法和返回方法，其典型代码如下：



在Builder（抽象类）中声明了一系列抽象的buildPartX()方法用于创建复杂产品的各个部件，具体建造过程在ConcreteBuilder（具体建造者）中实现，此外还提供了工厂方法getResult()，用于返回一个建造好的完整产品。

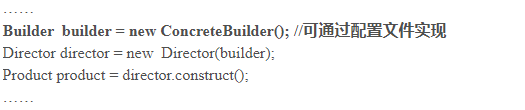
在ConcreteBuilder（具体建造者）中实现了buildPartX()方法，通过调用Product（产品角色）的setPartX()方法可以给产品对象的成员属性设值。不同的具体建造者在实现buildPartX()方法时将有所区别，如setPartX()方法的参数可能不一样，在有些具体建造者类中某些setPartX()方法无须实现（提供一个空实现）。而这些对于客户端来说都无须关心，客户端只需知道具体建造者类型即可。

      在建造者模式的结构中还引入了一个Director（指挥者类），该类主要有两个作用：一方面它隔离了客户与创建过程；另一方面它控制产品的创建过程，包括某个buildPartX()方法是否被调用以及多个buildPartX()方法调用的先后次序等。指挥者针对抽象建造者编程，客户端只需要知道具体建造者的类型，即可通过指挥者类调用建造者的相关方法，返回一个完整的产品对象。在实际生活中也存在类似指挥者一样的角色，如一个客户去购买电脑，电脑销售人员相当于指挥者，只要客户确定电脑的类型，电脑销售人员可以通知电脑组装人员给客户组装一台电脑。



在指挥者类中可以注入一个抽象建造者类型的对象，其核心在于提供了一个建造方法construct()，在该方法中调用了builder对象的构造部件的方法，最后返回一个产品对象。

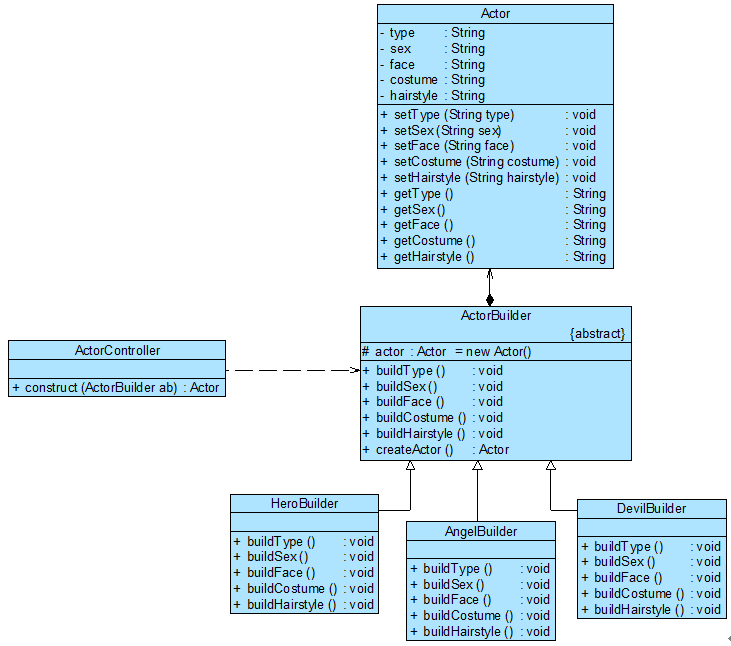
对于客户端而言，只需关心具体的建造者即可，一般情况下，客户端类代码片段如下所示：



可以通过配置文件来存储具体建造者类ConcreteBuilder的类名，使得更换新的建造者时无须修改源代码，系统扩展更为方便。在客户端代码中，无须关心产品对象的具体组装过程，只需指定具体建造者的类型即可。

建造者模式与抽象工厂模式有点相似，但是建造者模式返回一个完整的复杂产品，而抽象工厂模式返回一系列相关的产品；在抽象工厂模式中，客户端通过选择具体工厂来生成所需对象，而在建造者模式中，客户端通过指定具体建造者类型并指导Director类如何去生成对象，侧重于一步步构造一个复杂对象，然后将结果返回。如果将抽象工厂模式看成一个**汽车配件生产厂**，生成不同类型的汽车配件，那么建造者模式就是一个**汽车组装厂**，通过对配件进行组装返回一辆完整的汽车。

例子：使用建造者模式实现游戏角色的创建。基本结构如下图



角色描述

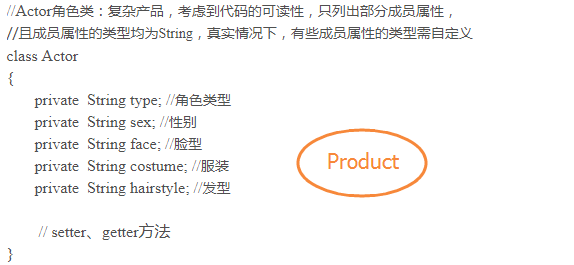
1. 复杂产品： Actor

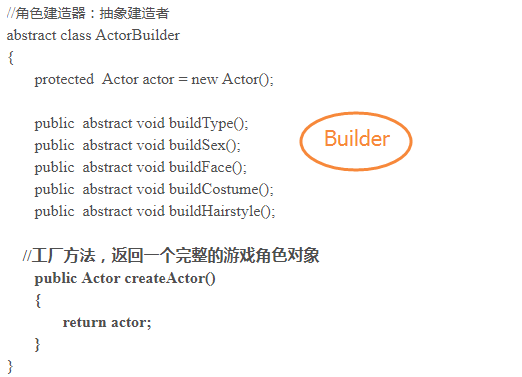
2. 指挥者： ActorController

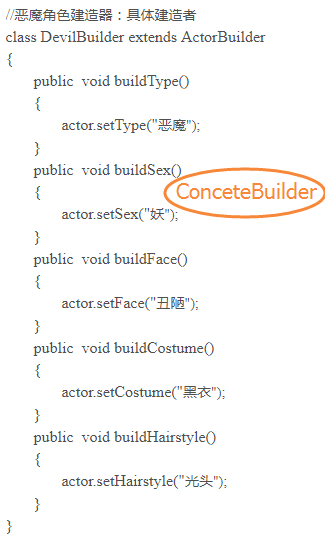
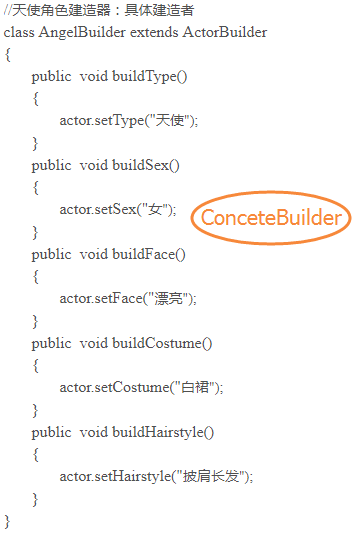
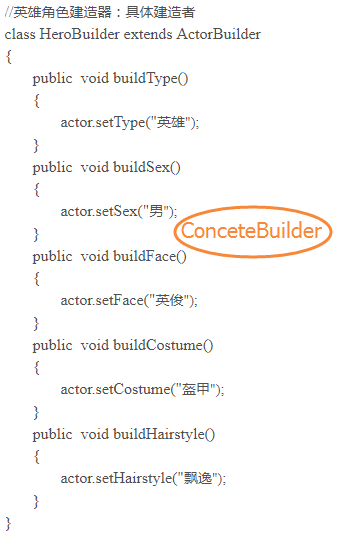
3. 抽象建造者： ActorBuilder

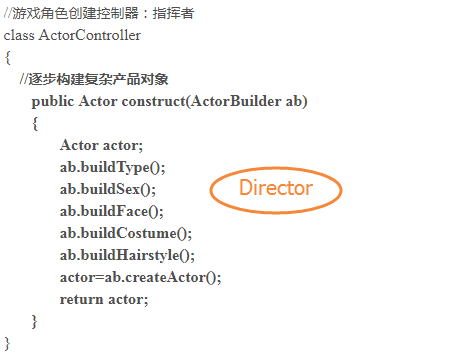
4. 具体建造者： HeroBuilder、AngelBuilder和DevilBuilder

代码实现：



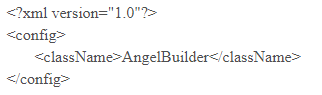




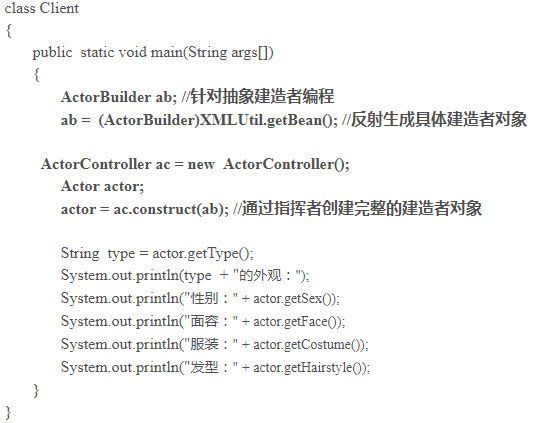


为了提高系统的灵活性和可扩展性，我们将具体建造者类的类名存储在配置文件中，并通过工具类XMLUtil来读取配置文件并反射生成对象。

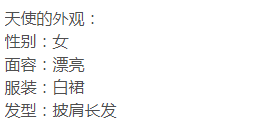
配置文件config.xml中存储了具体建造者类的类名，代码如下所示：



编写如下客户端测试代码：



编译并运行程序，输出结果如下：

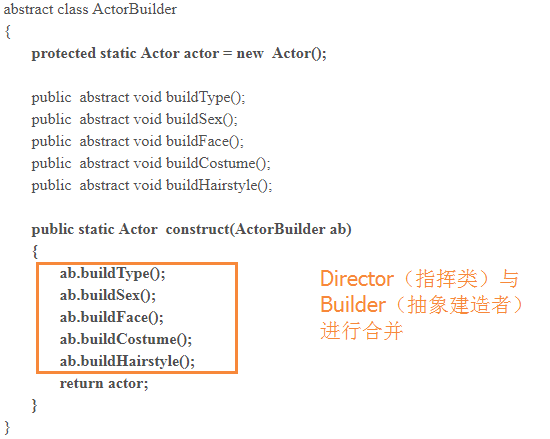


在建造者模式中，客户端只需实例化指挥者类，指挥者类针对抽象建造者编程，客户端根据需要传入具体的建造者类型，指挥者将指导具体建造者一步一步构造一个完整的产品（逐步调用具体建造者的buildX()方法），相同的构造过程可以创建完全不同的产品。在游戏角色实例中，如果需要更换角色，只需要修改配置文件，更换具体角色建造者类即可；如果需要增加新角色，可以增加一个新的具体角色建造者类作为抽象角色建造者的子类，再修改配置文件即可，原有代码无须修改，完全符合“开闭原则”。

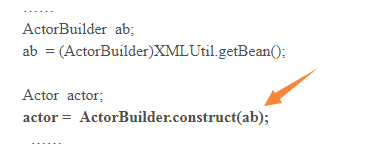
关于Director（指挥类）的进一步讨论

1. 省略Director

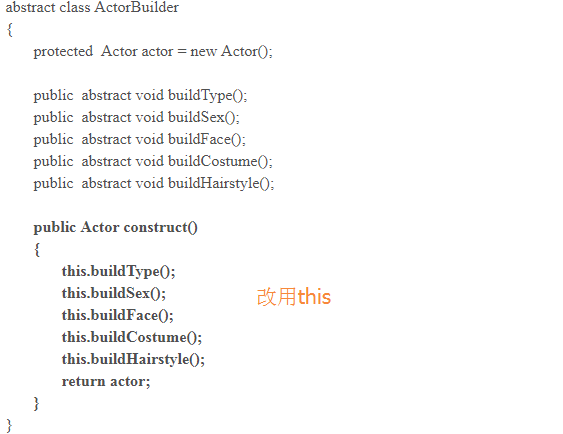
在有些情况下，为了简化系统结构，可以将Director和抽象建造者Builder进行合并，在Builder中提供逐步构建复杂产品对象的construct()方法。由于Builder类通常为抽象类，因此可以将construct()方法定义为静态(static)方法。如果将游戏角色设计中的指挥者类ActorController省略。



客户端代码紧接更改。



2. 省略方法参数



客户端调用抽象建造者（Builder）的construct()方法时则不需要再传递参数。

注意：

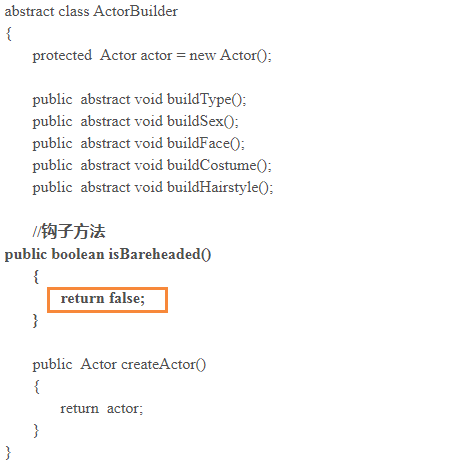
如果construct()方法较为复杂，待构建产品的组成部分较多，建议还是将construct()方法单独封装在Director中，这样做更符合“单一职责原则”。

3. 钩子方法的引入

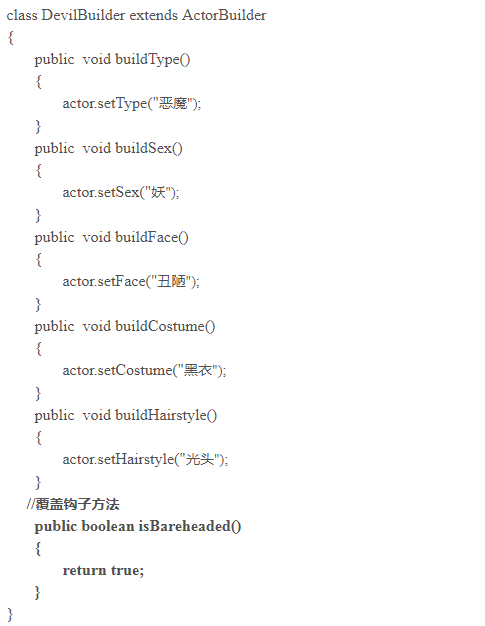
建造者模式除了逐步构建一个复杂产品对象外，还可以通过Director类来更加精细地控制产品的创建过程，例如增加一类称之为钩子方法(HookMethod)的特殊方法来**控制是否对某个buildPartX()的调用**。

钩子方法的返回类型通常为boolean类型，方法名一般为isXXX()，钩子方法定义在抽象建造者类中。

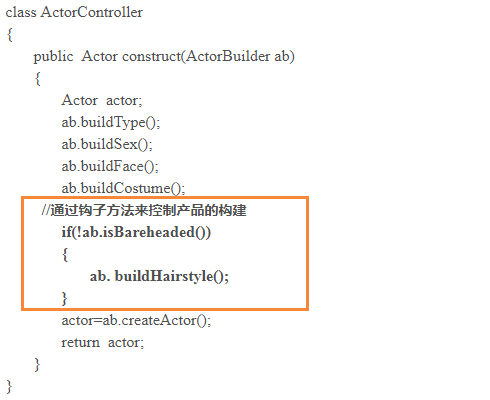
例如我们可以在游戏角色的抽象建造者类ActorBuilder中定义一个方法isBareheaded()，用于判断某个角色是否为“光头(Bareheaded)”，在ActorBuilder为之提供一个默认实现，其返回值为false，代码如下所示：



如果某个角色无须构建头发部件，例如“恶魔(Devil)”，则对应的具体建造器DevilBuilder将覆盖isBareheaded()方法，并将返回值改为true，代码如下所示：



此时，指挥者类ActorController的代码修改如下：



通过引入钩子方法，我们可以在Director中对复杂产品的构建进行精细的控制，不仅指定buildPartX()方法的执行顺序，还可以控制是否需要执行某个buildPartX()方法。

**建造者模式总结**

1. 主要优点：

(1) 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。

(2) 每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，用户使用不同的具体建造者即可得到不同的产品对象。由于指挥者类针对抽象建造者编程，增加新的具体建造者无须修改原有类库的代码，系统扩展方便，符合“开闭原则”

(3) 可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。

2. 主要缺点：

(1) 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，例如很多组成部分都不相同，不适合使用建造者模式，因此其使用范围受到一定的限制。

(2) 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大，增加系统的理解难度和运行成本。

3. 适用场景：

(1) 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性。

(2) 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。

(3) 对象的创建过程独立于创建该对象的类。在建造者模式中通过引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类和客户类中。

(4) 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品。

[建造者模式](https://blog.csdn.net/LoveLion/article/details/17517213)

[复杂对象的组装与创建——建造者模式（一）：游戏角色设计，建造者模式概述](https://blog.csdn.net/lovelion/article/details/7426015)

[复杂对象的组装与创建——建造者模式（二）：游戏角色设计的建造者模式解决方案](https://blog.csdn.net/lovelion/article/details/7426323)

[复杂对象的组装与创建——建造者模式（三）：关于Director的进一步讨论，建造者模式总结](https://blog.csdn.net/lovelion/article/details/7426855)